Semantics for Locking Specifications

Michael D. Ernst[1], Damiano Macedonio[2], Massimo Merro[3], and Fausto Spoto[2,3]

[1] Computer Science & Engineering, University of Washington, WA, USA
[2] Julia Srl, Verona, Italy
[3] Dipartimento di Informatica, Università degli Studi di Verona, Italy

Abstract. To prevent concurrency errors, programmers need to obey a locking discipline. Annotations that specify that discipline, such as Java's `@GuardedBy`, are already widely used. Unfortunately, their semantics is expressed informally and is consequently ambiguous. This article highlights such ambiguities and overcomes them by formalizing two possible semantics of `@GuardedBy`, using a reference operational semantics for a core calculus of a concurrent Java-like language. It also identifies when such annotations are actual guarantees against data races. Our work aids in understanding the annotations and supports the development of sound tools that verify or infer them.

1 Introduction

Concurrency can increase program performance by scheduling parallel independent tasks on multicore hardware, and can enable responsive user interfaces. However, concurrency might induce problems such as *data races*, *i.e.*, concurrent access to shared data by different threads, with consequent unpredictable or erroneous software behavior. Such errors are difficult to understand, diagnose, and reproduce. They are also difficult to prevent: testing tends to be incomplete due to nondeterministic scheduling choices made by the runtime, and model-checking scales poorly to real-world code.

The simplest approach to prevent data races is to follow a *locking discipline* while accessing shared data: always hold a given lock when accessing a datum. It is easy to violate the locking discipline, so tools that verify adherence to the discipline are desirable. These tools require a *specification language* to express the intended locking discipline. The focus of this paper is on the formal definition of a specification language, its semantics, and the guarantees that it gives against data races.

In Java, the most popular specification language for expressing a locking discipline is the `@GuardedBy` annotation. Informally, if the programmer annotates a field or variable f as `@GuardedBy(`E`)` then a thread may access f only while holding the monitor corresponding to the *guard expression* E. The `@GuardedBy` annotation was proposed by Goetz [12] as a documentation convention only, without tool support. The annotation has been adopted by practitioners; GitHub contains about 35,000 uses of the annotation in 7,000 files. Tool support now exists in Java PathFinder [18], the Checker Framework [10], IntelliJ [22], and Julia [24].

All of these tools rely on the previous informal definition of `@GuardedBy(`E`)` [16]. However, such an informal description is prone to many ambiguities. Suppose a field f is annotated as `@GuardedBy(`E`)`, for some guard expression E. (1) The definition above does not clarify how an occurrence of the special variable `this` [4] in E should be interpreted in client code; this actually depends on the context in which f is accessed. (2) It does not define what an *access* is. (3) It does not say whether the lock statement must use the guard expression E as written or whether a different expression that evaluates to the same value is permitted. (4) It does not indicate whether the lock that must be taken is E at the time of synchronization or E at the time of field access: side effects on E might make a difference here. (5) It does not clarify whether the lock on the guard E must be taken when accessing the field *named* f or the *value* bound to f.

The latter ambiguity is particularly important. The interpretation of `@GuardedBy` based on names is adopted in most tools appearing in the literature [18,22,24], whereas the interpretation based on values seems to be less common [10,24]. As a consequence, it is interesting to understand whether and how these two possible interpretations of `@GuardedBy` actually protect against data races on the annotated field/variable.

The main contribution of this article is the formalization of two different semantics for annotations of the form `@GuardedBy(`E`)` *`Type`* `x`: a *name-protection* semantics, in which accesses to the annotated *name* `x` need to be synchronized on the guard expression E, and a *value-protection* semantics, in which accesses to a *value* referenced by `x` need to be synchronized on E. The semantics clarify all the above ambiguities, so that programmers and tools know what those annotations mean and which guarantees they entail. We then show that both the name-protection and the value-protection semantics can protect against data races under proper restrictions on the variables occurring in the guard expression E. The name-protection semantics requires a further constraint — the protected variable or field must not be aliased. Our formalization relies on a reference semantics for a concurrent fragment of Java, which we provide in the *structural operational semantics* style of Plotkin [23].

We have used our formalization to extend the Julia static analyzer [24] to check and infer `@GuardedBy` annotations in arbitrary Java code. Julia allows the user to select either name-protection or value-protection. Our implementation reveals that most programmer-written `@GuardedBy` annotations do not satisfy either of the two alternative semantics given in this paper. For instance, in the code of Google Guava [13] (release 18), the programmer put 64 annotations on fields; 17 satisfy the semantics of name protection; 9 satisfy the semantics of value protection; the others do no satisfy any of the two. Fig. 1 shows an example of an annotation written by the programmers of Guava and that statisfy only the name protection. Namely, field `runnables` is annotated as `@GuardedBy(this)` but its value is accessed without synchronization at line `140` [13]. In this extended abstract proofs are omitted; they can be found in the appendix.

[4] Normally, `this` denotes the Java reference to the current object.

Fig. 1 Guava 18's `com.google.common.util.concurrent.ExecutionList` class. The `@GuardedBy` annotation (line 66) is satisfied for name, but not for value, protection.

```
47  public final class ExecutionList {
66    private @GuardedBy(this) RunnableExecutorPair runnables;
116   public void execute() {
117     RunnableExecutorPair list;
118     synchronized (this) {
123       list = runnables;
124       runnables = null;
125     }
137     RunnableExecutorPair reversedList = null;
138     while (list != null) {
139       RunnableExecutorPair tmp = list;
140       list = list.next;
141       tmp.next = reversedList;
142       reversedList = tmp;
143     }
148   }
177 }
```

Outline. Sec. 2 discusses the informal semantics of `@GuardedBy` by way of examples. Sec. 3 defines the syntax and semantics of a concurrent fragment of Java. Sec. 4 gives formal definitions for both the name-protection and value-protection semantics. Sec. 5 shows which guarantees they provide against data races. Sec. 6 describes the implementation in Julia. Sec. 7 discusses related work and concludes.

2 Informal Semantics of `@GuardedBy`

This section illustrates the use of `@GuardedBy` by example. Fig. 2 defines an observable object that allows clients to concurrently register listeners. Registration must be synchronized to avoid data races: simultaneous modifications of the `ArrayList` might result in a corrupted list or lost registrations. Synchronization is needed in the `getListeners()` method as well, or otherwise the Java memory model does not guarantee the inter-thread visibility of the registrations.

The interpretation of the `@GuardedBy(this)` annotation on field `listeners` requires resolving the ambiguities explained in Sec. 1. The intended locking discipline is that every use of `listeners` should be enclosed within a construct `synchronized (`*container*`) {...}`, where *container* denotes the object whose field `listeners` is accessed (ambiguities (1) and (2)). For instance, the access `original.listeners` in the copy constructor is enclosed within `synchronized (original) {...}`. This contextualization *of the guard expression*, similar to viewpoint adaptation [11], is not clarified in any informal definitions of `@GuardedBy` (ambiguity (3)). Furthermore, it is not clear if a definite alias of `original` can be used as synchronization guard at line 5. It is not clear if `original` would be allowed to be reassigned between lines 5 and 6 (ambiguity (4)). Note that the copy constructor does not synchronize on `this` even though it accesses `this.listeners`. This is safe so long as the constructor does not leak `this`. This paper assumes that an escape analy-

Fig. 2 This code has a potential data race due to aliasing of the `listeners` field.

```
public class Observable {
  private @GuardedBy(this) List<Listener> listeners = new ArrayList<>();
  public Observable() {}
  public Observable(Observable original) { // copy constructor
    synchronized (original) {
      listeners.addAll(original.listeners);
    }
  }
  public void register(Listener listener) {
    synchronized (this) {
      listeners.add(listener);
    }
  }
  public List<Listener> getListeners() {
    synchronized (this) {
      return listeners;
    }
  }
}
```

sis [6] has established that constructors do not leak `this`. The `@GuardedBy(this)` annotation on field `listeners` suffers also from ambiguity (5): it is not obvious whether it intends to protect the *name* `listeners` (*i.e.*, the name can be only used when the lock is held) or the value currently bound to `listeners` (*i.e.*, that value can be only accessed when the lock is held). Another way of stating this is that `@GuardedBy` can be interpreted as a *declaration annotation* (a restriction on uses of a name) or as a *type annotation* (a restriction on values associated to that name).

The code in Fig. 2 seems to satisfy the name-protection locking discipline expressed by the annotation `@GuardedBy(this)` for field `listeners`: every use of `listeners` occurs in a program point where the current thread locks its container, and we conclude that `@GuardedBy(this)` name-protects `listeners`. Nevertheless, a data race is possible, since two threads could call `getListeners()` and later access the returned value concurrently. This is inevitable when critical sections *leak* guarded data. More generally, name protection does not prevent data races if there are aliases of the guarded name (such as a returned value in our example) that can be used in an unprotected manner. The value-protection semantics of `@GuardedBy` is not affected by aliasing as it tracks accesses to the value referenced by the name, not the name itself.

Any formal definition of `@GuardedBy` must result in mutual exclusion in order to ban data races. If x is `@GuardedBy(`E`)`, then at every program point P where a thread accesses x (or its value), the thread must hold the lock on E. Mutual exclusion requires that two conditions are satisfied: (i) E can be evaluated at all program points P, and (ii) these evaluations always yield the same value.

Point (i) is syntactic and related to the fact that E cannot refer to variables or fields that are not always in scope or visible at all program points P. This problem exists for both name protection and value protection, but is more significant for the latter, that is meant to protect values that flow in the program through arbitrary aliasing. For instance, the annotation `@GuardedBy(listeners)` cannot be used for value protection in Fig. 2, since the name `listeners` is not

Fig. 3 Value protection prevents data races; see `itself` in the guard expression.

```
public class Observable {
  private @GuardedBy(itself) List<Listener> listeners = new ArrayList<>();
  public Observable() {}
  public Observable(Observable original) { // copy constructor
    synchronized (original.listeners) {
      listeners.addAll(original.listeners);
    }
  }
  public void register(Listener listener) {
    synchronized (listeners) {
      listeners.add(listener);
    }
  }
  public List<Listener> getListeners() {
    synchronized (listeners) {
      return listeners;
    }
  }
}
```

visible outside class `Observable`, but its value flows outside that class through method `getListeners()` and must be protected also if it accessed there. The value protection semantics supports a special variable `itself` in E, that refers to the current value of x being protected, without problems of scope or visibility. For instance, for value protection, the code in Fig. 2 could be rewritten as in Fig. 3.

Point (ii) is semantical and related to the intent of providing a guarantee of mutual exclusion. For instance, in Fig. 3, value protection bans data races on `listeners` since the guard `itself` can be evaluated everywhere (point (i)) and always yields the value of `listeners` itself (point (ii)). Here, the `@GuardedBy(itself)` annotation requires all accesses to the value of `listeners` to occur only when the current thread locks the same monitor — even outside class `Observable`, in a client that operates on the value returned by `getListeners()`. In Fig. 4, instead, field `listeners` is `@GuardedBy(guard)` according to both name protection and value protection, but the value of `guard` is distinct at different program points: no mutual exclusion guarantee exists and data races on `listeners` occur.

3 A Core Calculus for Concurrent Java

Some preliminary notions are needed to define our calculus. A *partial function* f from A to B is denoted by $f : A \rightharpoonup B$, and its *domain* is $\mathrm{dom}(f)$. We write $f(v)\downarrow$ if $v \in \mathrm{dom}(f)$ and $f(v)\uparrow$ otherwise. The symbol ϕ denotes the empty function, such that $\mathrm{dom}(\phi) = \emptyset$; $\{v_1 \mapsto t_1, \ldots, v_n \mapsto t_n\}$ denotes the function f with $\mathrm{dom}(f) = \{v_1, \ldots, v_n\}$ and $f(v_i) = t_i$ for $i = 1, \ldots, n$; $f[v_1 \mapsto t_1, \ldots, v_n \mapsto t_n]$ denotes the update of f, where $\mathrm{dom}(f)$ is enlarged for every i such that $v_i \notin \mathrm{dom}(f)$. A tuple is denoted as $\langle v_0, \ldots, v_n\rangle$. A *poset* is a structure $\langle A, \leq\rangle$ where A is a set with a reflexive, transitive, and antisymmetric relation $\leq$. Given $a \in A$, we define $\uparrow a \stackrel{def}{=} \{a' : a \leq a'\}$. A *chain* is a totally ordered poset.

Fig. 4 If the guard expression refers to distinct values at distinct program points, concurrent accesses to `listeners` can race.

```
public class Observable {
  private @GuardedBy(guard) List<Listener> listeners = new ArrayList<>();
  private Object guard1 = new Object();
  private Object guard2 = new Object();
  public Observable() {}
  public Observable(Observable original) { // copy constructor
    Object guard = guard1;
    synchronized (guard) {
      listeners.addAll(original.listeners);
    }
  }
  public void register(Listener listener) {
    Object guard = guard2;
    synchronized (guard) {
      listeners.add(listener);
    }
  }
}
```

3.1 Syntax

Symbols $f, g, x, y, \ldots$ range over a set of variables *Var* that includes `this`. Variables identify either local variables in methods or instance variables (*fields*) of objects. Symbols $m, p, \ldots$ range over a set *MethodName* of method names. There is a set *Loc* of memory locations, ranged over by l. Symbols $\kappa, \kappa_0, \kappa_1, \ldots$ range over a set of *classes* (or *types*) *Class*, ordered by a *subclass relation* $\leq$; $\langle \mathit{Class}, \leq \rangle$ is a poset such that for all $\kappa \in \mathit{Class}$ the set $\uparrow\kappa$ is a finite chain. Intuitively, $\kappa_1 \leq \kappa_2$ means that κ_1 is a *subclass* (or *subtype*) of κ_2. If $m \in \mathit{MethodName}$, then $\kappa.m$ denotes the implementation of m inside class κ, if any. The partial function $\mathit{lookup}(\,) : \mathit{Class} \times \mathit{MethodName} \rightharpoonup \mathit{Class}$ formalizes *Java's dynamic method lookup*, *i.e.* the runtime process of determining the class containing the implementation of a method on the basis of the class of the receiver object: $\mathit{lookup}(\kappa, m) \stackrel{\mathit{def}}{=} \mathit{min}(\uparrow \kappa.m)$ if $\uparrow \kappa.m \neq \emptyset$ and is undefined otherwise, where $\uparrow\kappa.m \stackrel{\mathit{def}}{=} \{\kappa' \in \uparrow\kappa \mid m \text{ is implemented in } \kappa'\}$ is a finite chain since $\uparrow\kappa.m \subseteq \uparrow\kappa$.

The set of *expressions* *Exp*, ranged over by E, and the set of *commands* *Com*, ranged over by C, are defined as follows. *Method bodies*, ranged over by B, are `skip`-terminated commands.

$$
\begin{array}{lcl}
E & ::= & x \;\mid\; E.f \;\mid\; \kappa\langle f_1 = E_1, \ldots, f_n = E_n \rangle \\
C & ::= & \texttt{decl}\ x = E \;\mid\; x := E \;\mid\; x.f := E \;\mid\; C; C \;\mid\; \texttt{skip} \;\mid\; E.m(\,) \;\mid \\
 & & \texttt{spawn}\ E.m(\,) \;\mid\; \texttt{sync}(E)\{C\} \;\mid\; \texttt{monitor\_enter}(l) \;\mid\; \texttt{monitor\_exit}(l) \\
B & ::= & \texttt{skip} \;\mid\; C; \texttt{skip}
\end{array}
$$

Constructs of our language are simplified versions of those of Java. For instance, loops must be implemented through recursion. We assume that the compiler ensures some standard syntactical properties, such as the same variable cannot be declared twice in a method, and the only free variable in a method's body is `this`. These simplifying assumptions can be relaxed without affecting our results.

Fig. 5 Running example.

```
public class K {
  private K1 x = new K1();
  private K2 y = new K2();
  public void m() {
    K1 z = x;
    K2 w = new Object();
    synchronized (z) {
      y = z.f;
      w = y;
    }
    w.g = new Object();
  }
}

class K1 {
  K2 f = new K2();
}

class K2 {
  Object g = new Object();
}
```

	by-name	by-value
field `x`	–	`@GuardedBy(itself)`
field `y`	`@GuardedBy(this.x)`	–
variable `z`	`@GuardedBy(itself)`	`@GuardedBy(itself)`
variable `w`	–	–

Expressions are variables, field accesses, and a construct for object creation, $\kappa\langle f_1 = E_1, \ldots, f_n = E_n\rangle$, that creates an object of class κ and initializes each field f_i to the value of E_i. Command `decl` declares a local variable. The declaration of a local variable in the body B of a method m must introduce a *fresh* variable never declared before in B, whose lifespan starts from there and reaches the end of B. The commands for variable/field assignment, sequential composition, and termination are standard. Method call $E.m(\,)$ looks up and runs method m on the runtime value of E. Command `spawn` $E.m(\,)$ does the same asynchronously, on a new thread. Command `sync(`E`){`C`}` is like Java's `synchronized`: the command C can be executed only when the current thread holds the lock on the value of E. `monitor_enter(`l`)` and `monitor_exit(`l`)` cannot be used by the programmer: our semantics introduces them in order to implement object synchronization.

The set of classes is $\mathit{Class} \stackrel{\mathit{def}}{=} \{\kappa : \mathit{MethodNames} \rightharpoonup B \mid \mathrm{dom}(\kappa) \text{ is finite}\}$. The binding of fields to their defining class is not relevant in our formalization. Given a class κ and a method name m, if $\kappa(m) = B$ then κ implements m with body B. For simplicity, `this` is the only free variable in B and methods have no formal parameters and/or return value. A *program* is a finite set of classes and includes a distinguished class *Main* that only defines a method *main* where the program starts: $\mathit{Main} \stackrel{\mathit{def}}{=} \{\mathit{main} \;\mapsto\; B_{\mathit{main}}\}$.

Example 1. Fig. 5 gives our *running example* in Java. In our core language, the body of method `m` is translated as follows: $B_{\mathtt{m}}$ = `decl z = this.x; decl w = Object`$\langle\rangle$`; sync(z) { this.y := z.f; w := this.y }; w.g := Object`$\langle\rangle$`; skip`, with classes `K` $\stackrel{\mathit{def}}{=} \{\mathtt{m} \;\mapsto\; B_{\mathtt{m}}\}$, `K1` $\stackrel{\mathit{def}}{=} \phi$, `K2` $\stackrel{\mathit{def}}{=} \phi$, and `Object` $\stackrel{\mathit{def}}{=} \phi$.

3.2 Semantic Domains

A running program consists of a pool of threads that share a memory. Initially, a single thread runs the main method. The `spawn` $E.m(\,)$ command adds a new

thread to the existing ones. Each thread has an activation stack S and a set $\mathcal{L}$ of locations that it currently locks. The activation stack S is a stack of activation records R of methods. Each R consists of the identifier $\kappa.m$ of the method, the command C to be executed when R will be on top of the stack (*continuation*), and the *environment* or binding σ that provides values to the variables in scope in R. For simplicity, we only have classes and no primitive types, so the only possible *values* are locations. Formally, $\mathit{Env} \stackrel{\mathit{def}}{=} \{\sigma : \mathit{Var} \rightharpoonup \mathit{Loc} \mid \mathrm{dom}(\sigma) \text{ is finite}\}$.

Definition 1. Activation records*, ranged over by R,* activation stacks*, ranged over by S, and* thread pools*, ranged over by T, are defined as follows:*

$$
\begin{array}{rcll}
R & ::= & \kappa.m[C]_\sigma & \textit{(activation record for } \kappa.m\textit{)}\\
S & ::= & \varepsilon \;\;\big|\;\; R :: S & \textit{(activation stack, possibly empty)}\\
T & ::= & \lceil S \rceil \mathcal{L} \;\;\big|\;\; T \parallel T & \textit{(thread pool).}
\end{array}
$$

The number of threads in T is written as $\#T$.

An object o is a triple containing the object's class, an environment binding its fields to their corresponding values, and a lock, *i.e.*, an integer counter incremented whenever a thread locks the object (locks are re-entrant). A *memory* μ maps a finite set of already allocated memory locations into *objects*.

Definition 2. Objects *and* memories *are defined as $\mathit{Object} \stackrel{\mathit{def}}{=} \mathit{Class} \times \mathit{Env} \times \mathbb{N}$ and $\mathit{Memory} \stackrel{\mathit{def}}{=} \{\mu : \mathit{Loc} \rightharpoonup \mathit{Object} \mid \mathrm{dom}(\mu)$ is finite$\}$, with selectors $\mathit{class}(o) \stackrel{\mathit{def}}{=} \kappa$, $\mathit{env}(o) \stackrel{\mathit{def}}{=} \sigma$ and $\mathit{lock}^{\#}(o) \stackrel{\mathit{def}}{=} n$, for every $o = \langle \kappa, \sigma, n \rangle \in \mathit{Object}$. We also define $o[f \mapsto l] \stackrel{\mathit{def}}{=} \langle \kappa, \sigma[f \mapsto l], n \rangle$ and $\mathit{lock}^{+}(o) \stackrel{\mathit{def}}{=} \langle \kappa, \sigma, n{+}1 \rangle$ and $\mathit{lock}^{-}(o) \stackrel{\mathit{def}}{=} \langle \kappa, \sigma, \max(0, n{-}1) \rangle$.*

For simplicity, we do not model delayed publication of field updates, allowed in the Java memory model, as that is not relevant for our semantics and results. Our goal is to identify expressions definitely locked at selected program points and locking operations are immediately published in the Java memory model. Hence, our memory model is a deterministic map shared by all threads.

The *evaluation of an expression E* in an environment σ and in a memory μ, written $[\![E]\!]^{\mu}_{\sigma}$, yields a pair $\langle l, \mu' \rangle$, where l is a location (the runtime value of E) and μ' is the memory resulting after the evaluation of E. Given a pair $\langle l, \mu \rangle$ we use selectors $\mathit{loc}(\langle l, \mu \rangle) = l$ and $\mathit{mem}(\langle l, \mu \rangle) = \mu$.

Definition 3 (Evaluation of Expressions). *The evaluation function has the type $[\![\;]\!] : (\mathit{Exp} \times \mathit{Env} \times \mathit{Memory}) \rightharpoonup (\mathit{Loc} \times \mathit{Memory})$ and is defined as:*

$$
[\![x]\!]^{\mu}_{\sigma} \stackrel{\mathrm{def}}{=} \langle \sigma(x), \mu \rangle \qquad [\![E.f]\!]^{\mu}_{\sigma} \stackrel{\mathrm{def}}{=} \langle \mathit{env}(\mu'(l))(f), \mu' \rangle, \text{ where } [\![E]\!]^{\mu}_{\sigma} = \langle l, \mu' \rangle
$$

$$
[\![\kappa\langle f_1{=}E_1, .., f_n{=}E_n \rangle]\!]^{\mu}_{\sigma} \stackrel{\mathrm{def}}{=} \langle l, \mu_n[l \mapsto \langle \kappa, \sigma', 0 \rangle] \rangle, \text{ where}
$$

(1) $\mu_0 = \mu$ *and* $\langle l_i, \mu_i \rangle = [\![E_i]\!]^{\mu_{i-1}}_{\sigma}$, *for* $i \in [1..n]$

(2) l *is* fresh *in* μ_n*, that is* $\mu_n(l)\uparrow$

(3) $\sigma' \in \mathit{Env}$ *is such that* $\sigma'(f_i) = l_i$ *for* $i \in [1..n]$*, while* $\sigma'(y)\uparrow$ *elsewhere.*

We assume that $[\![\;]\!]$ is undefined if any of the function applications is undefined.

In the evaluation of the object creation expression, a fresh location l is allocated and bound to an unlocked object whose environment σ' binds its fields to the values of the corresponding initialization expressions.

Table 1 Structural operational semantics for sequential commands.

$$\frac{[\![E]\!]^{\mu}_{\sigma} = \langle l,\, \mu' \rangle \quad \sigma(x)\uparrow \quad \sigma' \stackrel{def}{=} \sigma[x \mapsto l]}{\langle \lceil \kappa.m[\mathtt{decl}\ x = E]_{\sigma} \rceil \mathcal{L},\, \mu \rangle \xrightarrow{1} \langle \lceil \kappa.m[\mathtt{skip}]_{\sigma'} \rceil \mathcal{L},\, \mu' \rangle} \text{ [decl]}$$

$$\frac{[\![E]\!]^{\mu}_{\sigma} = \langle l,\, \mu' \rangle \quad \sigma(x)\downarrow \quad \sigma' \stackrel{def}{=} \sigma[x \mapsto l]}{\langle \lceil \kappa.m[x := E]_{\sigma} \rceil \mathcal{L},\, \mu \rangle \xrightarrow{1} \langle \lceil \kappa.m[\mathtt{skip}]_{\sigma'} \rceil \mathcal{L},\, \mu' \rangle} \text{ [var-ass]}$$

$$\frac{[\![E]\!]^{\mu}_{\sigma} = \langle l,\, \mu' \rangle \quad o = \mu(\sigma(x)) \quad o' \stackrel{def}{=} o[f \mapsto l] \quad \mu'' \stackrel{def}{=} \mu'[\sigma(x) \mapsto o']}{\langle \lceil \kappa.m[x.f := E]_{\sigma} \rceil \mathcal{L},\, \mu \rangle \xrightarrow{1} \langle \lceil \kappa.m[\mathtt{skip}]_{\sigma} \rceil \mathcal{L},\, \mu'' \rangle} \text{ [field-ass]}$$

$$\frac{\langle \lceil \kappa.m[C_1]_{\sigma} \rceil \mathcal{L},\, \mu \rangle \xrightarrow{1} \langle \lceil \kappa.m[C_1']_{\sigma'} \rceil \mathcal{L}',\, \mu' \rangle \quad C_1 \neq E.p() \quad C_1 \neq \mathtt{spawn}\ E.p()}{\langle \lceil \kappa.m[C_1; C_2]_{\sigma} \rceil \mathcal{L},\, \mu \rangle \xrightarrow{1} \langle \lceil \kappa.m[C_1'; C_2]_{\sigma'} \rceil \mathcal{L}',\, \mu' \rangle} \text{ [seq]}$$

$$\frac{-}{\langle \lceil \kappa.m[\mathtt{skip}; C]_{\sigma} \rceil \mathcal{L},\, \mu \rangle \xrightarrow{1} \langle \lceil \kappa.m[C]_{\sigma} \rceil \mathcal{L},\, \mu \rangle} \text{ [seq-skip]}$$

$$\frac{[\![E]\!]^{\mu}_{\sigma} = \langle l,\, \mu' \rangle \quad \kappa' = \mathit{lookup}(\mathit{class}(\mu'(l)), p) \quad \kappa'(p) = B}{\langle \lceil \kappa.m[E.p(\,); C]_{\sigma} :: S \rceil \mathcal{L},\, \mu \rangle \xrightarrow{1} \langle \lceil \kappa'.p[B]_{\{\mathtt{this} \mapsto l\}} :: \kappa.m[C]_{\sigma} :: S \rceil \mathcal{L},\, \mu' \rangle} \text{ [invoc]}$$

3.3 Structural Operational Semantics

Our operational semantics is given in terms of a *reduction relation* on *configurations* of the form $\langle T,\, \mu \rangle$, where T is a pool of threads and μ is a memory that models the heap of the system. We write $\langle T,\, \mu \rangle \xrightarrow{n} \langle T',\, \mu' \rangle$ for representing an execution step in which $n \geq 1$ denotes the position of the thread in T that fires the transition, starting from the leftmost in the pool T (thread 1). We write $\rightarrow$ instead of $\xrightarrow{n}$ to abstract on the running thread; $\rightarrow^*$ denotes the reflexive and transitive closure of $\rightarrow$. We first introduce reduction rules where the activation stack consists of a single activation record, then lift to the general case.

Table 1 deals with sequential commands. In rule [decl] an undefined variable x is declared. Rules [var-ass] and [field-ass] formalize variable and field assignment, respectively. Rule [seq] assumes that the first command is not of the form $E.p(\,)$ or `spawn` $E.p(\,)$; these two cases are treated separately. In rule [invoc] the receiver E is evaluated and the method implementation is looked up from the dynamic class of the receiver. The body of the method is put on top of the activation stack and is executed from an initial state where only variable `this` is in scope, bound to the receiver. Unlike previous rules, this rule deals with the whole activation stack rather than assuming only a single activation record.

Table 2 focuses on concurrency and synchronization. The spawn of a new method is similar to a method call, but the method body runs in its own new thread with an initially empty set of locked locations. In rule [sync] the location l associated to the guard E is computed; the computation can proceed only if a

Table 2 Structural operational semantics for concurrency and synchronization.

$$\frac{[\![E]\!]^{\mu}_{\sigma} = \langle l,\, \mu' \rangle \quad \kappa' = \mathit{lookup}(\mathit{class}(\mu'(l)), p) \quad \kappa'(p) = B}{\langle \lceil \kappa.m[\texttt{spawn}\ E.p(\,); C]_{\sigma} :: S \rceil \mathcal{L},\, \mu \rangle \xrightarrow{1} \langle \lceil \kappa'.p[B]_{\{\texttt{this} \mapsto l\}} :: \epsilon \rceil \emptyset \parallel \lceil \kappa.m[C]_{\sigma} :: S \rceil \mathcal{L},\, \mu' \rangle} \text{[spawn]}$$

$$\frac{[\![E]\!]^{\mu}_{\sigma} = \langle l, \mu' \rangle}{\langle \lceil \kappa.m[\texttt{sync(}E\texttt{)\{}C\texttt{\}}]_{\sigma} \rceil \mathcal{L},\, \mu \rangle \xrightarrow{1} \langle \lceil \kappa.m[\texttt{monitor\_enter(}l\texttt{)}; C; \texttt{monitor\_exit(}l\texttt{)}]_{\sigma} \rceil \mathcal{L},\, \mu' \rangle} \text{[sync]}$$

$$\frac{\mathit{lock}^{\#}(\mu(l)) = 0 \quad \mathcal{L}' \stackrel{\mathit{def}}{=} \mathcal{L} \cup \{l\} \quad \mu' \stackrel{\mathit{def}}{=} \mu[l \mapsto \mathit{lock}^{+}(\mu(l))]}{\langle \lceil \kappa.m[\texttt{monitor\_enter(}l\texttt{)}]_{\sigma} \rceil \mathcal{L},\, \mu \rangle \xrightarrow{1} \langle \lceil \kappa.m[\texttt{skip}]_{\sigma} \rceil \mathcal{L}',\, \mu' \rangle} \text{[acquire-lock]}$$

$$\frac{l \in \mathcal{L} \quad \mu' \stackrel{\mathit{def}}{=} \mu[l \mapsto \mathit{lock}^{+}(\mu(l))]}{\langle \lceil \kappa.m[\texttt{monitor\_enter(}l\texttt{)}]_{\sigma} \rceil \mathcal{L},\, \mu \rangle \xrightarrow{1} \langle \lceil \kappa.m[\texttt{skip}]_{\sigma} \rceil \mathcal{L},\, \mu' \rangle} \text{[reentrant-lock]}$$

$$\frac{\mathit{lock}^{\#}(\mu(l)) > 1 \quad \mu' \stackrel{\mathit{def}}{=} \mu[l \mapsto \mathit{lock}^{-}(\mu(l))]}{\langle \lceil \kappa.m[\texttt{monitor\_exit(}l\texttt{)}]_{\sigma} \rceil \mathcal{L},\, \mu \rangle \xrightarrow{1} \langle \lceil \kappa.m[\texttt{skip}]_{\sigma} \rceil \mathcal{L},\, \mu' \rangle} \text{[decrease-lock]}$$

$$\frac{\mathit{lock}^{\#}(\mu(l)) = 1 \quad \mathcal{L}' \stackrel{\mathit{def}}{=} \mathcal{L} \setminus \{l\} \quad \mu' \stackrel{\mathit{def}}{=} \mu[l \mapsto \mathit{lock}^{-}(\mu(l))]}{\langle \lceil \kappa.m[\texttt{monitor\_exit(}l\texttt{)}]_{\sigma} \rceil \mathcal{L},\, \mu \rangle \xrightarrow{1} \langle \lceil \kappa.m[\texttt{skip}]_{\sigma} \rceil \mathcal{L}',\, \mu' \rangle} \text{[release-lock]}$$

Table 3 Structural operational semantics: structural rules.

$$\frac{\langle \lceil R \rceil \mathcal{L},\, \mu \rangle \xrightarrow{1} \langle \lceil R' \rceil \mathcal{L}',\, \mu' \rangle}{\langle \lceil R{::}S \rceil \mathcal{L},\, \mu \rangle \xrightarrow{1} \langle \lceil R'{::}S \rceil \mathcal{L}',\, \mu' \rangle} \text{[push]} \qquad \frac{-}{\langle \lceil \kappa.m[\texttt{skip}]_{\sigma}{::}S \rceil \mathcal{L},\, \mu \rangle \xrightarrow{1} \langle \lceil S \rceil \mathcal{L},\, \mu \rangle} \text{[pop]}$$

$$\frac{\langle T_1,\, \mu \rangle \xrightarrow{n} \langle T_1',\, \mu' \rangle}{\langle T_1 \parallel T_2,\, \mu \rangle \xrightarrow{n} \langle T_1' \parallel T_2,\, \mu' \rangle} \text{[par-l]} \qquad \frac{\langle T_2,\, \mu \rangle \xrightarrow{n} \langle T_2',\, \mu' \rangle}{\langle T_1 \parallel T_2,\, \mu \rangle \xrightarrow{\#T_1 + n} \langle T_1 \parallel T_2',\, \mu' \rangle} \text{[par-r]}$$

$$\frac{-}{\langle \lceil \varepsilon \rceil \mathcal{L} \parallel T,\, \mu \rangle \xrightarrow{1} \langle T,\, \mu \rangle} \text{[end-l]} \qquad \frac{-}{\langle T \parallel \lceil \varepsilon \rceil \mathcal{L},\, \mu \rangle \xrightarrow{\#T+1} \langle T,\, \mu \rangle} \text{[end-r]}$$

lock action is possible on l. The lock will be released only at the end of the critical section C. Rule [acquire-lock] models the entering of the monitor of an unlocked object. Rule [reentrant-lock] models Java's *lock reentrancy*. Rule [decrease-lock] decreases the lock counter of an object that still remains locked, as it was locked more than once. When the lock counter reaches 0, rule [release-lock] can fire to release the lock of the object.

In Table 3, rule [push] lifts the execution of an activation record to that of a stack of activation records. The remaining structural rules are straightforward.

Definition 4 (Operational Semantics of a Program). *The initial configuration of a program is* $\langle T_0,\, \mu_0 \rangle$ *where* $T_0 \stackrel{\mathit{def}}{=} \lceil \mathit{Main}.\mathit{main}[B_{\mathit{main}}]_{\{\texttt{this} \mapsto l_{\mathit{init}}\}} \rceil \emptyset$ *and* $\mu_0 \stackrel{\mathit{def}}{=} \{l_{\mathit{init}} \mapsto \langle \mathit{Main},\, \phi, 0 \rangle\}$. *The* operational semantics *of a program is the set of traces of the form* $\langle T_0,\, \mu_0 \rangle \;\; \rightarrow^* \;\; \langle T,\, \mu \rangle$.

Example 2. The implementation in Ex. 1, becomes a program by defining B_{main} as: $\texttt{K}\langle\, \texttt{x} = \texttt{K1}\langle \texttt{f} = \texttt{K2}\langle \texttt{g} = \texttt{Object}\langle\rangle\rangle\rangle,\ \texttt{y} = \texttt{K2}\langle \texttt{g} = \texttt{Object}\langle\rangle\rangle\rangle\texttt{.m}();\texttt{skip}$. The operational semantics builds the following maximal trace from $\langle T_0,\, \mu_0\rangle$ that, for convenience, we divide in eight macro-steps:

1. $\to^* \langle\lceil \texttt{K.m}[\texttt{decl z} = \texttt{this.x};\ldots]_{\sigma_1} :: \mathit{Main.main}[\texttt{skip}]_{\{\texttt{this}\mapsto l_{init}\}}\rceil\emptyset,\, \mu_1\rangle$
   with $\mu_1 \stackrel{def}{=} \mu_0[l \mapsto o, l_1 \mapsto o_1, l_2 \mapsto o_2, l_3 \mapsto o_3, l_4 \mapsto o_4, l_5 \mapsto o_4]$;
   $o \stackrel{def}{=} \langle \texttt{K},\, \{\texttt{x} \mapsto l_1, \texttt{y} \mapsto l_2\}, 0\rangle$; $o_1 \stackrel{def}{=} \langle \texttt{K1},\, \{\texttt{f} \mapsto l_3\}, 0\rangle$; $o_2 \stackrel{def}{=} \langle \texttt{K2},\, \{\texttt{g} \mapsto l_4\}, 0\rangle$;
   $o_3 \stackrel{def}{=} \langle \texttt{K2},\, \{\texttt{g} \mapsto l_5\}, 0\rangle$; $o_4 \stackrel{def}{=} \langle \texttt{Object},\, \phi, 0\rangle$; $\sigma_1 \stackrel{def}{=} \{\texttt{this} \mapsto l\}$
2. $\to^* \langle\lceil \texttt{K.m}[\texttt{decl w} = \texttt{Object}\langle\rangle;\ldots]_{\sigma_2} :: \ldots\rceil\emptyset,\, \mu_1\rangle$ with $\sigma_2 \stackrel{def}{=} \sigma_1[\texttt{z} \mapsto l_1]$
3. $\to^* \langle\lceil \texttt{K.m}[\texttt{sync(z)\{...\}};\ldots]_{\sigma_3} :: \ldots\rceil\emptyset,\, \mu_2\rangle$ with $\mu_2 \stackrel{def}{=} \mu_1[l_6 \mapsto o_4]$; $\sigma_3 \stackrel{def}{=} \sigma_2[\texttt{w} \mapsto l_6]$
4. $\to^* \langle\lceil \texttt{K.m}[\texttt{this.y} := \texttt{z.f};\ldots;\texttt{monitor\_exit}(l_1);\ldots]_{\sigma_3} :: \ldots\rceil\{l_1\},\, \mu_3\rangle$
   with $\mu_3 \stackrel{def}{=} \mu_2[l_1 \mapsto \mathit{lock}^+(o_1)]$
5. $\to^* \langle\lceil \texttt{K.m}[\texttt{w} := \texttt{this.y};\texttt{monitor\_exit}(l_1);\ldots]_{\sigma_3} :: \ldots\rceil\{l_1\},\, \mu_4\rangle$
   with $\mu_4 \stackrel{def}{=} \mu_3[l \mapsto o']$; $o' \stackrel{def}{=} \langle \texttt{K},\, \{\texttt{x} \mapsto l_1, \texttt{y} \mapsto l_3\}, 0\rangle$
6. $\to^* \langle\lceil \texttt{K.m}[\texttt{monitor\_exit}(l_1);\texttt{w.g} := \texttt{Object}\langle\rangle;\texttt{skip}]_{\sigma_4} :: \ldots\rceil\{l_1\},\, \mu_4\rangle$ with $\sigma_4 \stackrel{def}{=} \sigma_3[\texttt{w} \mapsto l_3]$
7. $\to^* \langle\lceil \texttt{K.m}[\texttt{w.g} := \texttt{Object}\langle\rangle;\texttt{skip}]_{\sigma_4} :: \ldots\rceil\emptyset,\, \mu_5\rangle$ with $\mu_5 \stackrel{def}{=} \mu_4[l_1 \mapsto o_1]$
8. $\to^* \langle\lceil \mathit{Main.main}[\texttt{skip}]_{\{\texttt{this}\mapsto l_{init}\}}\rceil\emptyset,\, \mu_6\rangle$ with $\mu_6 \stackrel{def}{=} \mu_5[l_5 \mapsto o_4]$.

Our semantics lets us formalize some properties on the soundness of the locking mechanism, that we report in Appendix B. Here we just report a key property used in our proofs, that states that two threads never lock the same location (*i.e.*, object) at the same time. It is proved by induction on the length of the trace.

Proposition 1. *Let* $\langle T_0,\, \mu_0\rangle \;\to^*\; \langle\lceil S_1\rceil\mathcal{L}_1 \parallel \ldots \parallel \lceil S_n\rceil\mathcal{L}_n\,,\, \mu\rangle$ *be an arbitrary trace. For any* $i, j \in \{1\ldots n\}$, $i \neq j$ *entails* $\mathcal{L}_i \cap \mathcal{L}_j = \emptyset$.

4 Two Semantics for `@GuardedBy` Annotations

This section gives two distinct formalizations for locking specifications of the form `@GuardedBy(`E`)` *`Type`* `x`, where E is any expression allowed by the language, possibly using a special variable `itself` that stands for the protected entity.

4.1 Name-Protection Semantics

In a *name-protection* interpretation, a thread must hold the lock on the value of the guard expression whenever it *accesses* (reads or writes) the *name* of the guarded variable/field. Def. 5 formalizes the notion of *accessing an expression* when a given command is executed. For our purposes, it is enough to consider a single execution step; thus the accesses in $C_1; C_2$ are only those in C_1. When an object is created, only its creating thread can access it. Thus field initialization cannot originate data races and is not considered as an access. The access refers to the value of the expression, not to its lock counter, hence `sync(`E`){`C`}` does not access E. For accesses to a field f, Def. 5 keeps the exact expression used for the container of f, that will be used in Def. 7 for the contextualization of `this`.

Definition 5 (Expressions Accessed in a Single Reduction Step). *The set of expressions accessed in a single execution step is defined as follows:*

$$
\begin{array}{rlrl}
\mathrm{acc}(x) & \stackrel{\mathit{def}}{=} \{x\} & \mathrm{acc}(E.f) & \stackrel{\mathit{def}}{=} \mathrm{acc}(E) \cup \{E.f\} \\
 & & \mathrm{acc}(\kappa\langle f_1{=}E_1, \ldots, f_n{=}E_n\rangle) & \stackrel{\mathit{def}}{=} \bigcup_{i=1}^{n} \mathrm{acc}(E_i) \\
\mathrm{acc}(\texttt{decl}\ x = E) & \stackrel{\mathit{def}}{=} \mathrm{acc}(E) & \mathrm{acc}(x := E) & \stackrel{\mathit{def}}{=} \mathrm{acc}(x) \cup \mathrm{acc}(E) \\
\mathrm{acc}(C_1; C_2) & \stackrel{\mathit{def}}{=} \mathrm{acc}(C_1) & \mathrm{acc}(x.f := E) & \stackrel{\mathit{def}}{=} \mathrm{acc}(x.f) \cup \mathrm{acc}(E) \\
\mathrm{acc}(E.m(\,)) & \stackrel{\mathit{def}}{=} \mathrm{acc}(E) & \mathrm{acc}(\texttt{spawn}\ E.m(\,)) & \stackrel{\mathit{def}}{=} \mathrm{acc}(E) \\
\mathrm{acc}(\texttt{monitor\_enter(}l\texttt{)}) & \stackrel{\mathit{def}}{=} \emptyset & \mathrm{acc}(\texttt{monitor\_exit(}l\texttt{)}) & \stackrel{\mathit{def}}{=} \emptyset \\
\mathrm{acc}(\texttt{sync(}E.f\texttt{)\{}C\texttt{\}}) & \stackrel{\mathit{def}}{=} \mathrm{acc}(E) & \mathrm{acc}(\texttt{sync(}x\texttt{)\{}C\texttt{\}}) & \stackrel{\mathit{def}}{=} \emptyset \ \stackrel{\mathit{def}}{=} \ \mathrm{acc}(\texttt{skip})
\end{array}
$$

$$
\mathrm{acc}(\texttt{sync(}\kappa\langle f_1 = E_1, \ldots, f_n = E_n\rangle\texttt{)\{}C\texttt{\}}) \stackrel{\mathit{def}}{=} \mathrm{acc}(\kappa\langle f_1{=}E_1, \ldots, f_n{=}E_n\rangle).
$$

We say that a command C accesses *a variable x if and only if $x \in \mathrm{acc}(C)$; we say that C* accesses *a field f if and only if $E.f \in \mathrm{acc}(C)$, for some expression E.*

We now define `@GuardedBy` for local variables (Def. 6) and for fields (Def. 7). In Sec. 2 we have already discussed the reasons for using the special variable `itself` in the guard expressions when working with a value-protection semantics. In the name-protection semantics, `itself` denotes just an alias of the accessed name: `@GuardedBy(itself)` *`Type`* `x` is the same as `@GuardedBy(x)` *`Type`* `x`.

Definition 6 (`@GuardedBy` for Local Variables). *A local variable x of a method $\kappa.m$ in a program is* name protected *by `@GuardedBy(`E`)` if and only if for every derivation $\langle T_0, \mu_0\rangle \ \rightarrow^* \ \langle T, \mu\rangle \xrightarrow{n} \cdots$ in which the n-th thread in T is $\lceil \kappa.m[C]_\sigma :: S\rceil\mathcal{L}$, whenever C accesses x we have loc $\left(\llbracket E\rrbracket^{\mu}_{\sigma[\mathtt{itself}\mapsto\sigma(x)]}\right) \in \mathcal{L}$.*

Example 3. In Ex. 2, variable `z` of `K.m` is name protected by `@GuardedBy(this.x)` since the name `z` is accessed at the macro-step 5 only, where $\llbracket\mathtt{this.x}\rrbracket^{\mu_3}_{\sigma_3[\mathtt{itself}\mapsto l_1]} = \langle l_1, \mu_3\rangle$; during those reductions, the current thread holds the lock on the object bound to l_1. According to Def. 5, macro-steps 2 and 3 do not contain accesses since they are declarations; macro-step 4 does not access `z` since it is a synchronization.

Definition 7 (`@GuardedBy` for Fields). *A field f in a program is* name protected *by `@GuardedBy(`E`)` if and only if for every trace $\langle T_0, \mu_0\rangle \ \rightarrow^* \ \langle T, \mu\rangle \xrightarrow{n} \ldots$ in which the n-th thread in T is $\lceil \kappa.m[C]_\sigma :: S\rceil\mathcal{L}$, whenever C accesses f, i.e. $E'.f \in \mathrm{acc}(C)$, for some E', with $\llbracket E'\rrbracket^{\mu}_{\sigma} = \langle l', \mu'\rangle$ and $l'' = env(\mu'(l'))f$, we have loc $\left(\llbracket E\rrbracket^{\mu'}_{\sigma[\mathtt{this}\mapsto l', \mathtt{itself}\mapsto l'']}\right) \in \mathcal{L}$.*

Notice that the guard expression E is evaluated in a memory μ' obtained by the evaluation of the container of f, that is E', and in an environment where the special variable `this` is bound to l', *i.e.* the evaluation of the container of f.

Remark 1. Def. 6 and 7 evaluate the guard E at those program points where x is accessed, in order to verify that its lock is held by the current thread. Hence E can only refer to `itself` and variables in scope at those points, and for its evaluation we must use the current environment σ. A similar observation holds for the corresponding definitions for the value-protection semantics in next section.

Example 4. In Ex. 2, field `y` is name protected by `@GuardedBy(this.x)`. It is accessed at macro-step 5, where $[\![\mathtt{this.x}]\!]^{\mu_3}_{\sigma_3[\mathtt{this}\mapsto l,\,\mathtt{itself}\mapsto l_1]} = \langle l_1,\, \mu_3\rangle$, and at macro-step 6, where $[\![\mathtt{this.x}]\!]^{\mu_4}_{\sigma_4[\mathtt{this}\mapsto l,\,\mathtt{itself}\mapsto l_1]} = \langle l_1,\, \mu_4\rangle$. In both cases, the active and only thread holds the lock on the object bound to l_1.

4.2 Value-Protection Semantics

An alternative semantics for `@GuardedBy` protects the values held in variables or fields rather than their name. In this *value-protection* semantics, a variable x is `@GuardedBy(`E`)` if wherever a thread dereferences a location l eventually bound to x, it holds the lock on the object obtained by evaluating E at that point. In object-oriented parlance, *dereferencing a location* l means accessing the object stored at l in order to read or write a field. In Java, accesses to the lock counter are synchronized at a low level and the class tag is immutable, hence their accesses cannot give rise to data races and are not relevant here. Dereferences (Def. 8) are very different from accesses (Def. 5). For instance, statement `v.f := w.g.h` accesses expressions `v`, `v.f`, `w`, `w.g` and `w.g.h` but dereferences only the locations held in `v`, `w` and `w.g`: locations bound to `v.f` and `w.g.h` are left untouched. Def. 8 formalizes the set of locations dereferenced by an expression or command to access some field and keeps track of the fact that the access is for reading ($\rightarrow$) or writing ($\leftarrow$) the field. Hence dereference tokens are $l.f\!\leftarrow$ or $l.f\!\rightarrow$, where l is a location and f is the name of the field that is accessed in the object held in l.

Definition 8 (Dereferenced Locations). *Given a memory μ and an environment σ, the dereferences in a single reduction are defined as follows:*

$$
\begin{array}{ll}
\mathrm{deref}(x)^\mu_\sigma \stackrel{def}{=} \emptyset & \mathrm{deref}(E.f)^\mu_\sigma \stackrel{def}{=} \{\mathit{loc}\,([\![E]\!]^\mu_\sigma)\,.f\!\rightarrow\} \cup \mathrm{deref}(E)^\mu_\sigma \\
\multicolumn{2}{c}{\mathrm{deref}(\kappa\langle f_1 = E_1, \ldots, f_n = E_n\rangle)^\mu_\sigma \stackrel{def}{=} \bigcup_{i=1}^n \mathrm{deref}(E_i)^\mu_\sigma} \\
\mathrm{deref}(\mathtt{decl}\ x = E)^\mu_\sigma \stackrel{def}{=} \mathrm{deref}(E)^\mu_\sigma & \mathrm{deref}(x := E)^\mu_\sigma \stackrel{def}{=} \mathrm{deref}(E)^\mu_\sigma \\
\mathrm{deref}(\mathtt{sync(}E\mathtt{)\{}C\mathtt{\}})^\mu_\sigma \stackrel{def}{=} \mathrm{deref}(E)^\mu_\sigma & \mathrm{deref}(C_1; C_2)^\mu_\sigma \stackrel{def}{=} \mathrm{deref}(C_1)^\mu_\sigma \\
\mathrm{deref}(\mathtt{monitor\_enter(}l\mathtt{)})^\mu_\sigma \stackrel{def}{=} \emptyset & \mathrm{deref}(x.f := E)^\mu_\sigma \stackrel{def}{=} \{\sigma(x).f\!\leftarrow\} \cup \mathrm{deref}(E)^\mu_\sigma \\
\mathrm{deref}(\mathtt{monitor\_exit(}l\mathtt{)})^\mu_\sigma \stackrel{def}{=} \emptyset & \mathrm{deref}(\mathtt{skip})^\mu_\sigma \stackrel{def}{=} \emptyset \\
\multicolumn{2}{c}{\mathrm{deref}(E.m(\,))^\mu_\sigma \stackrel{def}{=} \mathrm{deref}(\mathtt{spawn}\ E.m(\,))^\mu_\sigma \stackrel{def}{=} \mathrm{deref}(E)^\mu_\sigma\ .}
\end{array}
$$

Its projection on locations is $\mathit{derefloc}(C)^\mu_\sigma = \{l \mid$ *there is f such that* $l.f\!\leftarrow\, \in \mathrm{deref}(C)^\mu_\sigma$ *or* $l.f\!\rightarrow\, \in \mathrm{deref}(C)^\mu_\sigma\}$.

Def. 9 fixes an arbitrary execution trace t and collects the set $\mathcal{L}$ of locations that have ever been bound to x in t. Then, it requires that whenever a thread dereferences one of those locations, that thread must hold the lock on the object obtained by evaluating the guard E.

Definition 9 (`@GuardedBy` for Local Variables). *A local variable x of a method $\kappa.m$ in a program is* value-protected *by* `@GuardedBy(`E`)` *if and only if for any derivation* $\langle T_0,\, \mu_0\rangle \xrightarrow{n_0} \cdots \xrightarrow{n_{i-1}} \langle T_i,\, \mu_i\rangle \xrightarrow{n_i} \ldots$, *letting*

- $T^n_j = \lceil k^n_j.m^n_j[C^n_j]_{\sigma^n_j} :: S^n_j\rceil \mathcal{L}^n_j$ *be the n-th thread of the pool T_j, for $j > 0$*
- $\mathcal{L} = \bigcup_{j>0}\{\sigma^{n_j}_j(x) \mid k^{n_j}_j.m^{n_j}_j = \kappa.m$ *and* $\sigma^{n_j}_j(x)\downarrow\}$ *be the set of locations eventually associated to variable x*

– $\mathcal{X}_i = \mathit{derefloc}(C_i^{n_i})^{\mu_i}_{\sigma_i^{n_i}} \cap \mathcal{L}$ *be those locations in* $\mathcal{L}$ *dereferenced at step* $\xrightarrow{n_i}$.

Then, for every $l \in \mathcal{X}_i$ *it follows that* $\mathit{loc}\left([\![E]\!]^{\mu_i}_{\sigma_i^{n_i}[\mathtt{itself}\mapsto l]}\right) \in \mathcal{L}_i^{n_i}$.

Note that $\mathcal{L}$ contains all locations eventually bound to x, also in the past, not just those bound to x in the last configuration $\langle T_i, \mu_i\rangle$. This is because the value of x might change during the execution of the program and flow through aliasing into other variables, that later get dereferenced.

Example 5. In Ex. 2 variable `z` is value-protected by `@GuardedBy(itself)`. The set $\mathcal{X}$ for `z` of Def. 9 is $\{l_1\}$. Location l_1 is only dereferenced at macro-step 5, where the corresponding object o_1 is accessed to obtain the value of its field f. At that program point, location l_1 is locked by the current thread.

Definition 10 (`@GuardedBy` for Fields). *A field f in a program is* value-protected *by* `@GuardedBy(`E`)` *if and only if for any derivation* $\langle T_0, \mu_0\rangle \xrightarrow{n_0} \cdots \xrightarrow{n_{i-1}} \langle T_i, \mu_i\rangle \xrightarrow{n_i} \cdots$ *, letting*

– $T_j^n = \lceil k_j^n.m_j^n[C_j^n]_{\sigma_j^n} :: S_j^n\rceil\mathcal{L}_j^n$ *be the* n*-th thread of the pool* T_j*, for* $j > 0$
– $\mathcal{L} = \bigcup_{j>0}\{\, \mathit{env}(\mu_j(l))(f) \mid l \in \mathrm{dom}(\mu_j) \text{ and } \mathit{env}(\mu_j(l))(f)\downarrow\}$ *be the set of locations eventually associated to field* f
– $\mathcal{X}_i = \mathit{derefloc}(C_i^{n_i})^{\mu_i}_{\sigma_i^{n_i}} \cap \mathcal{L}$ *be those locations in* $\mathcal{L}$ *dereferenced at step* $\xrightarrow{n_i}$.

Then, for every $l \in \mathcal{X}_i$ *it follows that* $\mathit{loc}\left([\![E]\!]^{\mu_i}_{\sigma_i^{n_i}[\mathtt{itself}\mapsto l]}\right) \in \mathcal{L}_i^{n_i}$.

Example 6. In Ex. 2 field `x` is value-protected by `@GuardedBy(itself)`. The set $\mathcal{X}$ for `x` of Def. 10 is $\{l_1\}$ and we conclude as in Ex. 5.

Remark 2. The two semantics for `@GuardedBy` are incomparable: neither entails the other. For instance, in Ex. 2 field `x` is value protected by `@GuardedBy(itself)`, but is not name protected: `x` is accessed at macro-step 1. Field `y` is name protected by `@GuardedBy(this.x)` but not value protected: its value is accessed at macro-step 8 via `w`. In some cases the two semantics do coincide. Variable `z` is `@GuardedBy(itself)` in both semantics: its name and value are only accessed at macro-step 5, where they are locked. Variable `w` is not `@GuardedBy(itself)` according to any semantics: its name and value are accessed at macro-step 8.

5 Protection against Data Races

In this section we provide sufficient conditions that ban data races when `@GuardedBy` annotations are satisfied, in either of the two versions of Sec. 4.1 and 4.2. First, we formalize the notion of *data race*. Informally, a data race occurs when two threads a and b dereference the same location l, at the same time, to access a field of the object stored at l and at least one of them, say a, modifies the field. We formalize below this definition for our language.

Definition 11 (Data race). *Let* $\langle T_0, \mu_0\rangle \rightarrow^* \langle T, \mu\rangle$, *where* $T_i = \lceil k_i.m_i[C_i]_{\sigma_i} :: S_i\rceil\mathcal{L}_i$ *is the* i*-th thread of* T*. A* data race *occurs at a location* l *at* $\langle T, \mu\rangle$ *during the access to a field* f *if there are* $a \neq b$ *such that* $\langle T, \mu\rangle \xrightarrow{a} \langle T', \mu'\rangle$, $\langle T, \mu\rangle \xrightarrow{b} \langle T'', \mu''\rangle$, $l.f{\leftarrow} \in \mathrm{deref}(C_a)^{\mu}_{\sigma_a}$ *and* ($l.f{\leftarrow} \in \mathrm{deref}(C_b)^{\mu}_{\sigma_b}$ *or* $l.f{\rightarrow} \in \mathrm{deref}(C_b)^{\mu}_{\sigma_b}$*).*

In Sec. 2 we said that accesses to variables (and fields) that are `@GuardedBy(`E`)` occur in mutual exclusion if the guard E is such that it can be evaluated at distinct program points and its evaluation always yields the same value. This means that E cannot contain local variables as they cannot be evaluated at distinct program points. Thus, we restrict the variables that can be used in E. In particular, `itself` can always be used since it refers to the location being dereferenced. For the name-protection semantics for fields, `this` can also be used, since it refers to the container of the guarded field, as long as it can be uniquely determined; for instance, if there is no aliasing. Indeed, Sec. 2 shows that name protection without aliasing restrictions does not ban data races, since it protects the name but not its value, that can be freely aliased and accessed through other names, without synchronization. In a real programming language, *aliasing* arises from assignments, returned values, and parameter passing. Our simple language has no returned values and only the implicit parameter `this`.

Definition 12 (Non-aliased variables and fields). *Let* P *be a program and* x *a variable or field name. We say that a name* x *is* non-aliased *in* P *if and only if for every arbitrary trace* $\langle T_0, \mu_0\rangle \rightarrow^* \langle T, \mu\rangle$ *of* P*, where* $T_i = \lceil k_i.m_i[C_i]_{\sigma_i} :: S_i\rceil\mathcal{L}_i$ *is the* i*-th thread of* T*, we have*

- *whenever* $\sigma_j(x) = l$*, for some* j *and* l*:*
  - *there is no* y*,* $y \neq x$*, such that* $\sigma_j(y) = l$
  - *there is no* k*,* $k \neq j$*, such that* $\sigma_k(y) = l$*, for some* y
  - *there is no* l' *such that* $env(\mu(l'))(y) = l$*, for some* y
- *whenever* $env(\mu(l'))(x) = l$*, for some* l' *and* l*:*
  - *there are no* y *and* j *such that* $\sigma_j(y) = l$
  - *there are no* y *and* l''*,* $l' \neq l''$*, such that* $env(\mu(l''))(y) = l$*.*

Checking if a name is non-aliased can be mechanized [3] and prevented by syntactic restrictions. Now, everything is in place to prove that, for non-aliased names, the name-protection semantics of `@GuardedBy` protects against data races.

Theorem 1 (Name-protection semantics vs. data race protection). *Let* E *be an expression in a program, and* x *be a* non-aliased *variable or field that is name protected by* `@GuardedBy(`E`)`*. If* x *is a variable, let* E *contain no variable distinct from* `itself`*; if* x *is a field, let* E *contain no variable distinct from* `itself` *and* `this`*. Then, no data race can occur at those locations bound to* x*, at any execution trace of that program.*

The absence of aliasing is not necessary for the value-protection semantics.

Theorem 2 (Value-protection semantics vs. data race protection). *Let* E *be an expression in a program, and* x *be a variable/field that is value-protected by* `@GuardedBy(`E`)`*. Let* E *have no variable distinct from* `itself`*. Then no data race can occur at those locations bound to* x*, during any execution of the program.*

Both results are proved by contradiction, by supposing that a data race occurs and showing that two threads would lock the same location, against Prop. 1

6 Implementation in Julia

The Julia static analyzer infers `@GuardedBy` annotations. The user selects the name-protection or the value-protection semantics. As discussed in Sec. 2, and then formalized in Sec. 4, a `@GuardedBy(`E`)` annotation holds for a variable or field x if, at all program points P where x is accessed (for name protection) or one of its locations is dereferenced (for value protection), the value of E is locked by the current thread. The inference algorithm of Julia builds on two phases: (i) compute P; (ii) find expressions E locked at all program points in P.

Point (i) is obvious for name protection, since accesses to x are syntactically apparent in the program. For value protection, the set P is instead undecidable, since there might be infinitely many objects potentially bound to x at runtime, that flow through aliasing. Hence Julia overapproximates P by abstracting objects into their *creation point* in the program: if two objects have distinct creation points, they must be distinct. The number of creation points is finite, hence the approximation is finitely computable. Julia implements creation points analysis as a concretization of the class analysis in [21], where objects are abstracted in their creation points instead of just their class tag.

Point (ii) uses the *definite aliasing* analysis of Julia, described in [19]. At each `synchronized(`G`)` statement, that analysis provides a set L of expressions that are definitely an alias of G at that statement (*i.e.*, their values coincide there, always). Julia concludes that the expressions in L are locked by the current thread after the `synchronized(`G`)` and until the end of its scope. Potential side-effects might however invalidate that conclusion, possibly due to concurrent threads. Hence, Julia only allows in L fields that are never modified after being defined, which can be inferred syntactically for a field. For name protection, viewpoint adaptation of `this` is performed on such expressions (Def. 7). These sets L are propagated in the program until they reach the points in P. The expressions E in point (ii) are hence those that belong to L at *all* program points in P.

Since `@GuardedBy(`E`)` annotations are expected to be used by client code, E should be visible to the client. For instance, Julia discards expressions E that refer to a private field or to a local variable that is not a parameter, since these would not be visible nor useful to a client.

The supporting creation points and definite aliasing analyses are sound, hence Julia soundly infers `@GuardedBy(`E`)` annotations that satisfy the formal definitions in Sec. 4. Such inferred annotations protect against data races if the sufficient conditions in Sec. 5 hold for them.

More detail and experiments with this implementation can be found in [**?**].

7 Conclusions, Future and Related Work

We have formalized two possible semantics for Java's `@GuardedBy` annotations. Coming back to the ambiguities sketched in Sec. 1, we have clarified that: (1) `this`

in the guard expression must be interpreted as the container of the guarded field and consistently contextualized (Def. 7). (2) An access is a variable/field use for name protection (Def. 5, 6, and 7). A value access is a dereference (field get/set or method call) for value protection; copying a value is not an access in this case (Def. 8, 9, and 10). (3) The value of the guard expression must be locked when a name or value is accessed, regardless of how it is accessed for locking (Def. 6, 7, 9, and 10). (4) The lock is taken on the value of the guard expression as evaluated at the access to the guarded variable or field (Def. 6, 7, 9, and 10 and rule [sync]). (5) Either the *name* or the *value* of a variable can be guarded, but this choice leads to very different semantics. Namely, in the *name-protection* semantics, the lock must be held whenever the variable/field's name is accessed (Def. 5, 6, and 7). In the *value-protection* semantics, the lock must be held whenever the variable/field's value is accessed (Def. 8, 9, and 10), regardless of what expression is used to access the value. Both semantics yield a guarantee against data races, though name protection requires an aliasing restriction (Th. 1 and 2).

This work could be extended by enlarging the set of guard expressions that protect against data races. In particular, we have found that programmers often use `this` in guard expressions, but in that case we have a proof of protection only for the name-protection semantics at the moment. Our simple language already admits local variables and global variables (in object-oriented languages, these are the fields of the objects). It could be further extended with static fields. We believe that the protection results in Sec. 5 still hold for them. Another aspect to investigate is the scope of the protection against data races. In this article, a single location is protected (Def. 11), not the whole tree of objects reachable from it: our protection is shallow rather than deep. Deep protection is possibly more interesting to the programmer, since it relates to a data structure as a whole, but it requires to reason about boundaries and encapsulation of data structures.

There are many other formalizations of the syntax and semantics of concurrent Java, such as [1,8]. There is a formalization that also includes extensions to Java such as RMI [2]. Our goal here is the semantics of annotations such as `@GuardedBy`. Hence we kept the semantics of the language to the minimum core needed for the formalization of those program annotations. Another well-known formalization is Featherweight Java [15], a functional language that provides a formal kernel of sequential Java. It does not include threads, nor assignment. Thus, it is not adequate to formalize data races, which need concurrency and assignments. Middleweight Java [5] is a richer language, with states, assignments and object identity. It is purely sequential, with no threads, and its formalization is otherwise at a level of detail that is unnecessarily complex for the present work. Welterweight Java [20] is a formalization of a kernel of Java that includes assignments to mutable data and threads. Our formalization is similar to theirs, but it is simpler since we do not model aspects that are not relevant to the definition of data races, such as subtyping. The need of a formal specification for reasoning about Java's concurrency and for building verification tools is recognized [9,17,7] but we are not aware of any formalization of the semantics of Java's concurrency annotations. Our formalization will support tools based on model-checking such as Java PathFinder [18] and Bandera [14,4], on type-checking such as the Checker Framework [10], or on abstract interpretation such as Julia [24]. Finally, our com-

panion paper [?] presents the details of the implementation of the Julia analyzer and of a type-checker for `@GuardedBy` annotations, together with extended experiments that show how these tools scale to large real software and provide useful results for programmers.

References

1. E. Ábrahám-Mumm, F. S. de Boer, W. P. de Roever, and M. Steffen. Verification for Java's Reentrant Multithreading Concept. In *FOSSACS*, pages 5–20, Grenoble, France, 2002.
2. A. Ahern and N. Yoshida. Formalising Java RMI with Explicit Code Mobility. *Theoretical Computer Science*, 389:341–410, 2007.
3. L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, DIKU, 1994.
4. Bandera. About Bandera. `http://bandera.projects.cis.ksu.edu`.
5. G. M. Bierman and M. J. Parkinson. Effects and Effect Inference for a Core Java Calculus. In *WOOD*, pages 82–107, 2003.
6. B. Blanchet. Escape Analysis for Java: and Practice. *ACM TOPLAS*, 25(6):713–775, 2003.
7. D. Bogdanas and G. Rosu. K-java: A Complete Semantics of Java. In *ACM SIGPLAN-SIGACT POPL*, pages 445–456, Mumbai, India, 2015.
8. P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. From Sequential to Multi-Threaded Java: An Event-Based Operational Semantics. In *AMAST*, pages 75–90, Sydney, AU, 1997.
9. J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. Expressing Checkable Properties of Dynamic Systems: the Bandera Specification Language. *STTT*, 4(1):34–56, 2002.
10. W. Dietl, S. Dietzel, M. D. Ernst, K. Muslu, and T. W. Schiller. Building and Using Pluggable Type-Checkers. In *ICSE*, pages 681–690, Waikiki, Honolulu, HI, USA, 2011.
11. W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In *ECOOP*, pages 28–53, Berlin, Germany, August 2007.
12. B. Goetz, T. Peierls, J. Bloch, and J. Bowbeer. *Java Concurrency in Practice*. Addison Wesley, May 2006.
13. Google. Guava: Google Core Libraries for Java 1.6+. `https://code.google.com/p/guava-libraries`.
14. J. Hatcliff and M. B. Dwyer. Using the Bandera Tool Set to Model-Check Properties of Concurrent Java Software. In *CONCUR*, pages 39–58, Aalborg, Denmark, August 2001.
15. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
16. Javadoc for `@GuardedBy`. `https://jsr-305.googlecode.com/svn/trunk/javadoc/javax/annotation/concurrent/Gu`
17. B. Long and B. W. Long. Formal Specification of Java Concurrency to Assist Software Verification. In *IPDPS*, page 136, Nice, France, April 2003. IEEE Computer Society.
18. NASA. Java PathFinder. `http://babelfish.arc.nasa.gov/trac/jpf`.
19. D. Nikolic and F. Spoto. Definite Expression Aliasing Analysis for Java Bytecode. In *ICTAC*, pages 74–89, Bangalore, India, September 2012.

20. J. Östlund and T. Wrigstad. Welterweight Java. In *TOOLS*, pages 97–116, Málaga, Spain, 2010.
21. J. Palsberg and M. I. Schwartzbach. Object-Oriented Type Inference. In *OOPSLA*, volume 26(11) of *ACM SIGPLAN Notices*, pages 146–161. ACM, November 1991.
22. V. Pech. Concurrency is Hot, Try the JCIP Annotations. `http://jetbrains.dzone.com/tips/concurrency-hot-try-jcip`, February 2010.
23. G.D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI-FN-19, Computer Science Department, Aarhus University, 1981.
24. Julia Srl. The Julia Static Analyzer. `http://www.juliasoft.com/julia`.

A Proofs of Sec 5

Theorem 3 (Name-protection semantics vs. data race protection). *Let E be an expression in a program, and x be a non-aliased variable or field that is name protected by* `@GuardedBy(`E`)`*. If x is a variable, let E contain no variable distinct from* `itself`*; if x is a field, let E contain no variable distinct from* `itself` *and* `this`*. Then, no data race can occur at those locations bound to x, at any execution trace of that program.*

Proof. The proof is by contradiction. Let $\langle T_0, \mu_0 \rangle \to^* \langle T, \mu \rangle$ be an arbitrary trace of our program, where $T_i = \lceil k_i.m_i[C_i]_{\sigma_i} :: S_i \rceil \mathcal{L}_i$ is the i-th thread of T. By Def. 11, if a data race occurred in $\langle T, \mu \rangle$, at some location l, bound to x, then $\langle T, \mu \rangle$ could evolve in at least two ways, say

$$\langle T, \mu \rangle \xrightarrow{a} \langle T', \mu' \rangle \text{ and } \langle T, \mu \rangle \xrightarrow{b} \langle T'', \mu'' \rangle$$

that dereference l in two different threads a and b. As x is non-aliased, by Def. 12 it cannot be used in one thread as a variable and in the other as a field. As $a \neq b$, by Def. 12 the name x cannot be used in both threads as local variable bound to the same location l. Thus, there is only one possibility: x is a field accessed by both threads, a and b, to dereference the location l. This means that there exist two expressions E_a and E_b such that $E_a.x \in \mathrm{acc}(C_a)$ and $E_b.x \in \mathrm{acc}(C_b)$. As x is non-aliased, by Def. 12 there cannot be two different containers (objects) of the same field x. As a consequence, the two expressions E_a and E_b must evaluate to the same value, *i.e.* $[\![E_a]\!]^{\mu}_{\sigma_a} = [\![E_b]\!]^{\mu}_{\sigma_b} = \langle l', \mu' \rangle$, for some l' and μ'. We recall that $\mathcal{L}_a$ and $\mathcal{L}_b$ denote the set of locations locked in $\langle T, \mu \rangle$ by thread a and b, respectively. As x is name protected by `@GuardedBy(`E`)`, Def. 7 entails that $loc\left([\![E]\!]^{\mu'}_{\sigma_a[\mathtt{this}\mapsto l'][\mathtt{itself}\mapsto l'']}\right) \in \mathcal{L}_a$ and $loc\left([\![E]\!]^{\mu'}_{\sigma_b[\mathtt{this}\mapsto l'][\mathtt{itself}\mapsto l'']}\right) \in \mathcal{L}_b$, for $l'' = env(\mu'(l'))(x)$. As x is a field, by hypothesis the guard expression E may only contain the variables `this` and `itself`. As a consequence, $loc\left([\![E]\!]^{\mu'}_{\sigma_a[\mathtt{this}\mapsto l'][\mathtt{itself}\mapsto l'']}\right) = loc\left([\![E]\!]^{\mu'}_{[\mathtt{this}\mapsto l'][\mathtt{itself}\mapsto l'']}\right) = loc\left([\![E]\!]^{\mu'}_{\sigma_b[\mathtt{this}\mapsto l'][\mathtt{itself}\mapsto l'']}\right)$. By Prop. 1 this is not possible as two threads cannot lock the same location at the same time.

The requirement on the absence of aliasing is not necessary when working with a value-protection semantics.

Theorem 4 (Value-protection semantics of `@GuardedBy` vs. data race protection). *Let E be an expression in a program, and x be a variable or field that is value-protected by* `@GuardedBy(`E`)`. *Let E contain no variable distinct from* `itself`. *Then no data race can occur at those locations bound to x, during any execution trace of the program.*

Proof. Again, the proof is by contradiction. Let $\langle T_0, \mu_0\rangle \xrightarrow{n_0} \cdots \xrightarrow{n_{i-1}} \langle T_i, \mu_i\rangle$ be an arbitrary trace of our program. If a data race occurred at a location l bound to x in that trace, then that trace could evolve in at least two ways, both dereferencing l but in distinct threads, say a and b (Def. 11). Since x is value protected by `@GuardedBy(`E`)`, both threads lock the value (*i.e.*, location) of E. Formally, by Def. 9 and 10 it follows that $loc\left(\llbracket E\rrbracket^{\mu_i}_{\sigma^a_i[\texttt{itself}\mapsto l]}\right) \in \mathcal{L}^a_i$ and $loc\left(\llbracket E\rrbracket^{\mu_i}_{\sigma^b_i[\texttt{itself}\mapsto l]}\right) \in \mathcal{L}^b_i$. Since `itself` is the only variable allowed in E, environments σ^a_i and σ^b_i are irrelevant in these evaluations and $loc\left(\llbracket E\rrbracket^{\mu_i}_{\sigma^a_i[\texttt{itself}\mapsto l]}\right) = loc\left(\llbracket E\rrbracket^{\mu_i}_{[\texttt{itself}\mapsto l]}\right) = loc\left(\llbracket E\rrbracket^{\mu_i}_{\sigma^b_i[\texttt{itself}\mapsto l]}\right)$. Again, by Prop. 1, this is impossible, as two threads cannot lock the same location at the same time.

B Properties of the Operational Semantics

Let us provide a few properties showing the soundness of both the locking and unlocking mechanisms of our operational semantics.

Two different threads never lock the same location:

Proposition 2 (Locking vs. multithreading). *Given an arbitrary execution trace*

$$\langle T_0, \mu_0\rangle \;\rightarrow^*\; \langle\lceil S_1\rceil\mathcal{L}_1 \parallel \ldots \parallel \lceil S_n\rceil\mathcal{L}_n\,,\, \mu\rangle$$

then for any $i, j \in \{1 \ldots n\}$, $i \neq j$ entails $\mathcal{L}_i \cap \mathcal{L}_j = \emptyset$.

Proof. By induction on the length of the trace.

When a thread starts its execution it does not hold any lock:

Proposition 3 (Thread initialization vs. locking). *Let*

$$\langle T_0, \mu_0\rangle \;\rightarrow^*\; \langle\lceil S_1\rceil\mathcal{L}_1 \parallel \ldots \parallel \lceil S_n\rceil\mathcal{L}_n\,,\, \mu\rangle \xrightarrow{i} \langle\lceil \hat{S}_1\rceil\hat{\mathcal{L}}_1 \parallel \ldots \parallel \lceil \hat{S}_m\rceil\hat{\mathcal{L}}_m\,,\, \hat{\mu}\rangle$$

be an arbitrary trace where $S_i = \kappa.m[\textbf{\textit{spawn}}\; E.p(\,); C]_\sigma :: S$, for some $\kappa, m, E, p, C, \sigma$ and S, then

- $\hat{\sigma}_i = \kappa'.p[B]_{\sigma'}$, *for appropriate κ', B and σ'*
- $\hat{\sigma}_{i+1} = \kappa.m[C]_\sigma :: S$
- $\hat{\mathcal{L}}_i = \emptyset$

– $\hat{\mathcal{L}}_{i+1} = \mathcal{L}_i$.

Proof. This is a direct conseuquence of the sematic rules [spawn], the only one which can be applied to perform the reduction step $\xrightarrow{i}$.

When a thread terminates it does not keep locks on locations:

Proposition 4 (Thread termination vs. locking). *Let*

$$\langle T_0,\, \mu_0\rangle \;\rightarrow^* \langle \lceil S_1 \rceil \mathcal{L}_1 \parallel \ldots \parallel \lceil S_n \rceil \mathcal{L}_n \,,\, \mu\rangle$$

be an arbitrary run where $S_i = \epsilon$, , *then* $\mathcal{L}_i = \emptyset$.

Proof. By induction on the lenght of the reduction.

A thread may not lock a location by mistake:

Proposition 5 (Locking). *Let*

$$\langle T_0,\, \mu_0\rangle \;\rightarrow^* \langle \lceil S_1 \rceil \mathcal{L}_1 \parallel \ldots \parallel \lceil S_n \rceil \mathcal{L}_n \,,\, \mu\rangle \xrightarrow{i} \langle \lceil \hat{S}_1 \rceil \hat{\mathcal{L}}_1 \parallel \ldots \parallel \lceil \hat{S}_m \rceil \hat{\mathcal{L}}_m \,,\, \hat{\mu}\rangle$$

be an arbitrary run. Then $\bigcup_{j=1}^{n} \mathcal{L}_j \subset \bigcup_{j=1}^{m} \hat{\mathcal{L}}_j$, *if and only if*

– $S_i = \kappa.m[\texttt{monitor\_enter(}l\texttt{)}; C']_\sigma :: S$, *for some* κ, m, l, C', σ *and* S
– $lock^{\#}(\mu(l)) = 0$ *and* $lock^{\#}(\hat{\mu}(l)) = 1$
– $\hat{\mathcal{L}}_i = \mathcal{L}_i \uplus \{l\}$
– $m = n$ *and* $\hat{\mathcal{L}}_j = \mathcal{L}_j$ *for every* $j \in \{1 \ldots n\} \setminus \{i\}$.

Reentrant locks are allowed: only threads that already own the lock on an object can synchronize again on that object.

Proposition 6 (Reentrant locking). *Given an arbitrary run*

$$\langle T_0,\, \mu_0\rangle \;\rightarrow^* \langle \lceil S_1 \rceil \mathcal{L}_1 \parallel \ldots \parallel \lceil S_n \rceil \mathcal{L}_n \,,\, \mu\rangle$$

where $l \in \bigcup_{j=1}^{n} \mathcal{L}_j$, *for some* l, *and* $S_i = \kappa.m[\texttt{monitor\_enter(}l\texttt{)}; C']_s :: S$, *for some* $i \in \{1..n\}$, $\kappa, m, C, E, C', \sigma$ *and* S. *Then*

$$\langle \lceil S_1 \rceil \mathcal{L}_1 \parallel \ldots \parallel \lceil S_n \rceil \mathcal{L}_n \,,\, \mu\rangle \xrightarrow{i} \langle \lceil \hat{S}_1 \rceil \hat{\mathcal{L}}_1 \parallel \ldots \parallel \lceil \hat{S}_n \rceil \hat{\mathcal{L}}_n \,,\, \hat{\mu}\rangle$$

if and only if

– $l \in \mathcal{L}_i$
– $lock^{\#}(\hat{\mu}(l)) = lock^{\#}(\mu(l)) + 1$
– $m = n$ *and* $\hat{\mathcal{L}}_j = \mathcal{L}_j$ *for every* $j \in \{1 \ldots n\}$.

Proof. By case analysis on the rule applied to perform the reduction. Here the only rule which can be applied is [reeentrant-lock].

Locks on locations are never released by mistake:

Proposition 7 (Lock releasing). *Let*

$$\langle T_0,\, \mu_0\rangle \;\rightarrow^* \langle \lceil S_1 \rceil \mathcal{L}_1 \parallel \ldots \parallel \lceil S_n \rceil \mathcal{L}_n \,,\, \mu\rangle \xrightarrow{i} \langle \lceil \hat{S}_1 \rceil \hat{\mathcal{L}}_1 \parallel \ldots \parallel \lceil \hat{S}_m \rceil \hat{\mathcal{L}}_m \,,\, \hat{\mu}\rangle$$

be an arbitrary run. Then $\bigcup_{j=1}^{n} \mathcal{L}_j \supset \bigcup_{j=1}^{m} \hat{\mathcal{L}}_j$*, if and only if*

- $S_i = \kappa.m[\mathtt{monitor\_exit}(l); C]_\sigma :: S$*, for some* κ, m, l, C, σ *and* S
- $\mathcal{L}_i = \hat{\mathcal{L}}_i \uplus \{l\}$
- $\mathit{lock}^\#(\mu(l)) = 1$ *and* $\mathit{lock}^\#(\hat{\mu}(l)) = 0$
- $m = n$ *and* $\hat{\mathcal{L}}_j = \mathcal{L}_j$ *for every* $j \in \{1 \ldots n\} \setminus \{i\}$.

Proof. By case analysis on the rule applied to perform the reduction. Here the only possible rule is [release-lock].

Unlocking always happens after some locking: it may release the lock or not, depending on the number of previous lockings.

Proposition 8 (Unlocking). *Let*

$$\langle T_0,\, \mu_0\rangle \;\rightarrow^* \langle \lceil S_1 \rceil \mathcal{L}_1 \parallel \ldots \parallel \lceil S_n \rceil \mathcal{L}_n \,,\, \mu\rangle \xrightarrow{i} \langle \lceil \hat{S}_1 \rceil \hat{\mathcal{L}}_1 \parallel \ldots \parallel \lceil \hat{S}_m \rceil \hat{\mathcal{L}}_m \,,\, \hat{\mu}\rangle$$

be an arbitrary run where $S_i = \kappa.m[\mathtt{monitor\_exit}(l); C]_\sigma :: S$*, for some* κ, m, C, σ *and* S*, then*

- $l \in \mathcal{L}_i$
- *if* $\mathit{lock}^\#(\mu(l)) > 1$ *then* $\hat{\mathcal{L}}_i = \mathcal{L}_i$ *else* $\mathcal{L}_i = \hat{\mathcal{L}}_i \uplus \{l\}$
- $\mathit{lock}^\#(\hat{\mu}(l)) = \mathit{lock}^\#(\mu(l)) - 1$
- $m = n$ *and* $\hat{\mathcal{L}}_j = \mathcal{L}_j$ *for every* $j \in \{1 \ldots n\} \setminus \{i\}$.

Proof. By case analysis on the rules applied to perform the reduction. Here there are two possible rules: [decrease-lock] and [release-lock].